Tema 1: Instrucciones básicas del programa Maple

Operaciones elementales en Maple

F5	Cambia la entrada de modo INPUT a TEXTO, permite escribir texto en la hoja
x + y;	SUMA. Las órdenes se terminan siempre con ; (salvo las que empiezan con ?)
x − y;	RESTA
x*y;	PRODUCTO (¡OJO! el por * siempre se escribe)
x / y;	DIVISION
x^y;	POTENCIA
a<b; a>b; a< > b; a<=b;	a menor que b, a mayor que b, a distinto de b, a menor o igual que b.
x:= 2;	ASIGNACION (la variable x toma a partir de ahora el valor 2)
x:= 'x';	DESASIGNACION (x vuelve a ser una variable) (apóstrofes, parte inferior tecla cierre de interrogación)
g:= x^2 + 1;	**EXPRESION** (asignamos a la variable g la expresión x^2 + 1)
f:=x->x^2 + 1;	CONSTRUCCION DE UNA **FUNCION**
if(0), f(2), f(−5);	Obtenemos los valores de f en 0, 2 y −5 (esto no puede hacerse con g)
Ctrl+k y Ctrl+j	Inserta entradas nuevas arriba o abajo
Shift + Enter	Pasa a la línea siguiente sin insertar una nueva entrada
Ctrl + Supr	Suprime una línea o grupo de ejecución

Interrupción e inicialización

Ctrl+Pausa	interrumpe los cálculos (también pulsar STOP de la barra de funciones)
restart;	Inicializa todas las variables globales y los path de librerías

Ordenes más frecuentes en Maple

?abs	petición de ayuda respecto de la orden abs
evalf(%);	calcula en punto flotante el resultado anterior
Digits:= 50;	ponemos a 50 el número de dígitos
evalf(Pi); evalf(Exp(1));	aproxima el valor de Pi y de E con 50 dígitos
Digits:= 10;	ponemos a 10 el número de dígitos
floor(−4.5);	función "parte entera" del Cálculo: entero inmediatamente inferior

?floor	recordemos que siempre podemos pedir ayuda de cualquier orden
abs(−3. 4);	Valor absoluto
100!;	**factorial** de 100
binomial(10, 3);	número combinatorio 10 sobre 3
gcd(8, 20);	máximo común divisor de dos números o polinomios
lcm(8, 12);	mínimo común múltiplo de dos números o polinomio
igcd(8, 20, 12);	máximo común divisor de cualesquiera números enteros
ilcm(8, 20, 12);	mínimo común múltiplo de cualesquiera números enteros
modp(8, 2);	resto de la división de 8 por 2
irem(m, n,' q')	calcula el resto de la división entera m entre n, y en q guarda el cociente (el argumento q es opcional, y se escribe entre apóstrofes, parte inferior tecla cierre de interrogación)
iquo(m, n,' r')	cociente de división entera m entre n, en r guarda el resto (es opcional)
ifactor(123456789);	Descomposición en factores primos de un entero
rand();	genera un entero no negativo aleatorio de 12 dígitos, ver vectores y matrices para una utilización particular
Nextprime(1000);	primo siguiente a 1000
rationalize(1/sqrt(2) + 1/sqrt(3));	racionaliza una expresión
ithprime(28);	devuelve el primo número 28 (2 es el primer nº primo)
isprime(7)	Test de primalidad

Operaciones con números complejos

s:= (2 + 4***I**) / (1 − 2***I**);	opera con complejos: I es la unidad imaginaria, asignamos el cociente a s
Re(s);	parte real del anterior cociente
Im(s);	parte imaginaria
argument(s);	devuelve el argumento principal: [−Pi, Pi]
abs(s);	módulo o valor absoluto si real

Asignaciones y substituciones

p:=x^2 + 4*x + 4;	asignamos el polinomio x^2+4*x+4 a la variable p
x:= 1;	hacemos x=1 y luego vemos cómo queda p
p;	es el valor del polinomio después de hacer x=1, si ahora tecleamos (x − 3)*(x − 4);
(x − 3)*(x − 4);	Escribirá 6, esta vía para averiguar cuanto vale p en x = 1 no es correcta, pues la x queda con el valor 1 para ulteriores cálculos
x:='x';	en primer lugar desasignamos x (**apóstrofes: parte inferior tecla cierre interrogación**)
(x − 3)*(x − 4);	ahora el polinomio queda correcto
subs(x = 1, p);	camino adecuado para averiguar cuanto vale una expresión al particularizar: utilizar subs
x;	no hemos asignamos nada a x y con la orden anterior sabemos cuanto vale p en x = 1

Operaciones con polinomios

x^2 + 3*x − 4;	¡No olvidemos!: el por * siempre se escribe
factor(%);	factorizamos la anterior expresión
expand(%);	Desarrollamos la anterior expresión
simplify(16^(1/2) − 3);	simplifica un cálculo, admite varias posibilidades, es aconsejable ?simplify;

Resolución de ecuaciones y sistemas

solve(x^2 + 3*x − 4, x);	resolvemos en x (con radicales)
fsolve(x^5 + 3*x^4 − 3*x^3 − 9*x^2 + 2*x + 6, x);	resolvemos en x aproximadamente (reales)
fsolve(x^5 + x^4 + 2*x^3 − x^2 + x − 1, x, complex);	resolución en x aproximada en punto flotante y complejos
solve({x + 2*y = 1,x − y = 3},{x, y});	resolución de sistemas
ecu:={x + y = a, x − 2*y = 1};	puede hacerse mediante asignaciones en más pasos: llamamos ecu al sistema
var:= {x, y};	var es el conjunto de variables
pepe:=**solve(ecu, var);**	la solución se le asigna a la variable pepe
linsolve(A, b);	resolución de un sistema lineal de matriz de coeficientes A y vector termino independiente b. Ver vectores y matrices para más información

Sumas y productos

1 + 3 + 5 + 7 + 9 + 11;	Expresaremos esta suma como:
sum(2*n − 1, n = 16);	para evitar que posibles asignaciones previas a la variable n produzcan errores es aconsejable escribirla como:
sum('2*n − 1','n' = 1 ..6);	Obsérvese que la variable y la expresión a sumar van entre apóstrofos (en la misma tecla que cierre de interrogación)
sum('k^2', 'k' = 1..n);	entre apóstrofos la k es una variable muda
product((x − k), k = 1..20);	expresión para el producto
expand(%);	desarrolla la anterior instrucción

Gráficas bidimensionales

plot(x^2, x = 0..2, title = `f(x) = x^2`);	dibuja la curva x^2 en el intervalo [0, 2], con título ¡Ojo el **título va entre alt-096**! (tecla [sin alterar seguida de espacio en blanco) ¡No confundir con el apóstrofo!
plot({x^2, x^3}, x = 0..1,);	dibujos simultáneos de las expresiones x^2 y x^3
plot([[−1, −1], [−1, 1], [1, 1], [1, −1], [−1, −1]], **style = point**);	dibujo de puntos con formato: (style = point) o (style = line) [[x_1, y_1], [x_2, y_2] ...]
plot([cos(2*t), t, t = 0..2*Pi], **coords = polar**);	dibuja curvas en polares

plot([sin(t), cos(t), t = 0..2*Pi]);	también en paramétricas
with(plots):	se accede al paquete gráfico, es necesario para las órdenes siguientes
polygonplot({[−1, 2], [3, 2], [−3, −2]}, color = red);	dibuja el polígono de vértices dados
Polygonplot ({[1, 1], [−3, 4], [7, 8], [−10, −11], \ [7, 9], [5, 8]}, color = green);	cuando una sentencia es muy larga para que quepa en una línea, al escribir un procedimiento ascii, se pone un **** (backslash) y se sigue en otra línea
p1:= **textplot**([−Pi/2, −1,`Minimo`]):	pone rótulos en puntos de coordenadas dadas, en este caso x = −Pi/2, y = −1 ¡Ojo a los dos puntos y al alt-096! Admite la opción: align = BELOW (o bien ABOVE, LEFT, RIGHT) hacer ?textplot para más detalles
p2:= plot(sin(x), x= −Pi..Pi):	almacena las instrucciones para dibujar la curva ¡Ojo a los dos puntos!
display([p1, p2]);	con esta orden representamos tanto p1 como p2, o cualquier otro número de rótulos
p3:= plot(cos(x), x = −Pi..Pi):	guardamos la gráfica del coseno para hacer un display.
display([p2, p3], insequence = true);	pinta de forma secuencial el seno y el coseno, y lo repite.
display([seq(plot(n*x, x = 0..3), n = −2..2)], insequence = true);	dibuja las rectas −2x, −x, 0, x, 2x y repite la secuencia
animate(n*x, x = 0..3, n = −2..2, **frames** = 20);	mueve las rectas anteriores de forma más continua
plot(f, −2..6); ó plot(f(x), x = −2..6);	dibuja la **función** f (f ha sido definida como función)
plot('f(x)', 'x' = −2..6); plot(f, −2..6);	deben emplearse cualquiera de estas dos instrucciones si f(x) es una **función o procedimiento** con algún **IF** en su interior. La primera de ellas va con apóstrofos

Taylor, límites, derivación e integración

p:= x/cos(x);	asignamos el cociente a la variable p
taylor(p, x = 0, 8);	obtenemos el desarrollo de Taylor del cociente en x = 0, con 8 términos. Obsérvese que no escribimos p(x) sino p a secas, si pusiéramos p(x) daría error, tampoco podemos escribir p(1) o p(6), sin embargo:
f:= x->sin(1/x);	construcción de una **función/procedimiento** (signo menos seguido de mayor que) imprescindible para poder utilizar f(−1), f(2), etc. ,..o en general f(x)
taylor(f(x), x = 1, 10);	desarrollo de Taylor de f(x) en x = 1 y con 10 términos, la utilización de f(x) requiere, la definición previa de f(x) como función/proc.
q1:= taylor(exp(x), x = 0, 3);	desarrollo de Taylor de E^x ,en x = 0, con 3 términos, que se asigna a la variable q1
eval(q1);	muestra el valor que hay en q1 (serie)

q2:= **convert(q1, polynom);**	convertimos q1 en polinomio y lo asignamos a la variable q2
plot({exp(x),q2(x)},x=-2..2);	comparación interesante: dibujamos la función y el polinomio de Taylor en x=0
limit((x^3 + x − 2) / (2*x^3 − x^2 + x + 4), x = infinity);	un límite cuando x tiende a infinito
limit(sin(x) / x, x = 0);	calculamos el límite cuando x tiende a 0
limit(1/(1+exp(−1/x)), x = 0, right);	Límite por la derecha cuando x tiende a 0
limit(1/(1+exp(−1/x)), x = 0, left);	Límite por la izquierda cuando x tiende a 0
diff(x^x, x);	derivamos x^x respecto de x
x$3;	sucesión de tres x, para utilizar en el cálculo de la derivada de orden tres
diff(x*cos(x), **x$3**);	derivada tercera de x*cos(x),
diff(diff(x^y, x), y);	derivada de x^y respecto de x, y luego el resultado respecto de y
int(x^3*cos(x), x);	integral de x^3*cos(x) respecto de x
int(x^2, x = a..b);	integral definida de x^2 entre los límites a y b
evalf(int(exp(−x^2), x = 0..1));	integración aproximada de E^(−x^2) entre los límites 0 y 1

Formas no ejecutables de diff, int y lim, sum, product

Diff(x^3, x);	expresión formal de la derivada **sin** calcular
Int(x^3, x = 0..1);	idem (solo con propósitos tipográficos)
Limit((x^3 + x) / (3*x^3 − 1), x = infinity);	idem (solo con propósitos tipográficos)
Sum((n^2 + n + 1) / (2*n^2 + 1), n = 1..infinity);	idem (solo con propósitos tipográficos)
Product(n, n = 1..30);	Idem

Cadenas de caracteres (strings)

dos:= `fichero7.m`;	son caracteres entre alt - 096, que asignamos a la variable dos
uno:= `c: /programas/`;	que pueden utilizarse como nombres (tecla [sin alterar seguida de blanco)
camino:= uno. dos camino:= cat(uno, dos);	concatenación de dos cadenas con el . , se utiliza en ocasiones para hacer read(camino); ver entrada y salida de proc.
nombre:= cat(`fichero`, s, `.m`);	también pueden concatenarse de esta forma, aquí s:= convert(n, string), n suponemos que es un entero. Podríamos introducir las anteriores sentencias en un bucle que variase n de 0 a 9 , y almacenar distintas gráficas o salidas con diferentes nombres, ver otras salidas gráficas.

Sucesiones

s:=−2, 5, 6, 0, −1;	las sucesiones son objetos maple
t:= seq(j^2, j = 1..10);	10 términos de la sucesión j^2 desde j = 1 hasta j=10

par:= seq(a + (b − a)*k / 10, k = 0..10);	Partición del intervalo [a, b] en 10 subintervalos de igual longitud: 11 puntos
plot({seq(E^(-n*x),n=1..5)}, x = −1..1);	son particularmente útiles en la representación de sucesiones de funciones.
s:= NULL;	Sucesión nula (¡Ojo NULL con mayúsculas!)

Conjuntos

a:= {perros, gatos, liebres, conejos};	los conjuntos también son objetos maple, su peculiaridad: el orden no importa y no hay elementos repetidos
b:= {perros, liebres};	conjunto b
a **intersect** b;	hay órdenes para trabajar con conjuntos, por ejemplo: la intersección
a **union** b;	unión
a **minus** b ;	diferencia
member(perros, a);	devolverá true o false dependiendo de si perros es o no elemento de a
B:= {sin(x), x^2, exp(x)};	pueden construirse conjuntos de funciones
plot(B, x = −1..1);	que luego pueden representarse simultáneamente
a:={ };	conjunto vacío

Listas

pepe:= **[**x, 1, y, −2, 0, 4**];**	listas: el orden importa
pepe[3]; ó **op**(3, pepe); op(2..4, pepe);	se accede a los elementos por índice, pero no se permiten asignaciones a pepe[3]. Se accede a los elementos del dos al cuatro
pepe:= **[op(pepe), z];**	añade un elemento nuevo la salida será [x, 1, y, −2, 0, 4, z]
nops(pepe);	se obtiene el nº de elementos de pepe.
pepe:=[ ];	lista nula
a:= **convert(a, list);**	un conjunto se convierte en lista
a:= **convert(a, set);**	una lista se convierte en conjunto
subsop(4 = **y**,[a, b, c, d, e, g]);	substitución en listas, la salida de esta instrucción será [a, b, c, y, e, g]
subsop(4= **NULL**, [a, b, c, d, e, g]);	el resultado ahora es eliminar la 4 entrada de la lista y queda [a, b, c, e, g]

Arrays y tablas

array(1..4);	array unidimensional vacío de 4 componentes **equivale a: vector(4)**
array(1..4, [−3, −2, 2, 3]); **array**([−3, −2, 2, 3]);	array unidimensional de 4 componentes, es equivalente a: vector(4, [−3, −2, 2, 3])
array(1..3, 1..3, [[1, 1, 0], [1, 1, 1], [0, 1, 1]]); **array**([[1, 1, 0], [1, 1, 1], [0, 1, 1]]);	array bidimensional de 3x3, **equivale a matrix**(3, 3, [[1, 1, 0], [1, 1, 1], [0, 1, 1]])

array(1..2, 1..2, 1..2);	array de 3 índices, en total 2x2x2 = 8 posiciones de almacenamiento, sin embargo, las matrices tienen más funciones asociadas como puede verse en el siguiente epígrafe
traduce:= **table**([rojo = (red, rot, rouge),\ azul =(blue, blau, bleu)]);	las tablas se diferencian de los array en que sus índices no son necesariamente enteros. Aquí asignamos dos sucesiones a los índices rojo y azul, podríamos asignar matrices ver ?table;y ?array; para más detalles
traduce[rojo];	nos devolverá : red, rot, rouge

Vectores y matrices

with(linalg):	accedemos al *package* de álgebra lineal. Hay de muy distintos tipos para listarlos ejecutar: ?packages;
u:= **vector**([−1, 3, 4, 7, 2]);	un vector se construye directamente o bien dando una regla para la formación de sus componentes
g:= i->i^2 + i;	construimos una función g
v:= vector(5, g);	construye un vector v de 5 componentes empezando en g(1)
vectdim(v);	devuelve el número de componentes del vector v
linalg[vectdim](v);	No hace falta ejecutar with(linalg) para utilizar vectdim, puede usarse como aparece a la izquierda. Especialmente indicada, para no tener que cargar path, desde procedimientos
v[3];	accedemos a la tercera componente mediante índice entre corchetes
v[2]:= 8;	no hay inconveniente en asignar valores a variables indexadas. Esa es la diferencia entre vectores y listas, para las listas hay que utilizar la incomoda subsop
h:= (i, j)->(1/(i + j − 1));	construimos una función de dos variables
A:= **matrix**(3, 3, h);	construimos una matriz de 3x3 mediante la anterior función h, empezando por h(1, 1)
rowdim(A);	devuelve el número de filas de la matriz A
row(A,i);	devuelve la fila i − ésima de A
coldim(A);	devuelve el número de columnas de la matriz A
col(A, j);	devuelve la columna j de la matriz A
det(A);	calcula el determinante de A
transpose(A);	matriz traspuesta de la dada
htranspose(A);	traspuesta conjugada de A
A[3,2];	accedemos al elemento (3, 2) de A
submatrix(A, 2..3, 1..2);	submatriz (a_{2, 1},a_{2, 2}\\a_{3, 1},a_{3, 2}) de A
B:= **array**([[1, −1, 0], [3, 2, 1], [1, 2, 3]]);	construcción directa de una matriz por asignación simple
C:= **multiply**(A, B);	Le asignamos a la variable C el producto de A por B
evalm(C);	en matrices hay que utilizar evalm para ver el resultado de la operación
b:= array([1, 0, −1, 0]);	B es un vector
ff:= **rand**(1..5):	La función ff generará números aleatorios entre 1 y

	5 .¡Ojo a los dos puntos, para que **no** muestre el resultado! Es así como se utiliza rand en este contexto.
A:= matrix(4, 4, ff);	esta es una matriz de 4*4 rellena con números aleatorios entre 1 y 5
linsolve(A, b);	resolvemos el sistema lineal A x = b
inverse(A);	nos da la inversa de A
evalf(eigenvals(A));	obtenemos los autovalores de A

Otras salidas gráficas distintas de pantalla

interface(**plotoutput** = `plot1. ps`);	para mandar la salida gráfica a un fichero que se llamará plot1.ps
interface(**plotdevice** = **ps**);	cambia dispositivo gráfico para que el anterior fichero salga en formato de impresora **postcript**
interface(**plotoptions** = **`portrait, noborder`**);	escogemos las opciones: dibujo vertical y sin enmarcar
interface(**plotoutput** = **`plot1.gif `**);	para mandar la salida gráfica a un fichero que se llamará plot1.gif
interface(**plotdevice** = **gif**);	cambia dispositivo gráfico para que el anterior fichero salga en formato **.gif** , que se podrá posteriormente visualizar con utilidades externas adecuadas.
interface(**plotdevice** = **win**);	restauramos el dispositivo a windows
interface(**plotoutput** = **terminal**);	restauramos la salida al terminal
interface(**plotoptions** = **default**);	restauramos a las opciones por defecto

Más sobre funciones

(ln**@**sin)(x);	composición de funciones = ln(sin(x))
(ln**@@3**)(x);	composición de funciones = ln(ln(ln(x)))
V:=**[seq(**−1 + 2*k/8, k = 0..8**)];**	creación de una lista [v_{1}, v_{2}, ..., v_{9}]
F:= x->1/(1 + 25*x^2);	construcción de una función
U:= **map(f,v);**	aplicamos la función f a los elementos de la lista v = [v_{1}, v_{2}, .., v_{9}], el resultado es la lista [f(v_{1}), f(v_{2}), ..., f(v_{9})]
F:= **unapply**(diff(x^x, x), x);	creamos una función a partir de evaluaciones
alias(sen = sin);	sen(x) equivale a la función sin(x)
diff(x*sen(x), x);	vemos que funciona exactamente como sin(x)
plot(sen(x), x = −Pi..Pi);	en todos los sentidos
F:=<x^2 + x + 1**|x>;**	una forma diferente de definir una función

Las variables time y lasterror

T0:= time( );	almacenamos el contenido de la variable tiempo del sistema en t0;
int(x^4*sin(x), x);	calculamos una primitiva
T1:= time( );	volvemos a almacenar tiempo en t1, t1-t0, nos dará aproximadamente el tiempo de cómputo de la integral
lprint(` Tiempo de cómputo	se utiliza desde dentro de un procedimiento para

=`, t1 − t0);	rotular posibles salidas, rótulos entre alt − 096

Manejo de expresiones y polinomios

collect(y*(sin(x) + 1) + sin(x), **sin(x));**	saca sin(x) factor común
expand(ln(x*a));	ln(x*a) pasa a ser ln(x) + ln(a)
combine(ln(x) + ln(a), ln);	ln(x) + ln(a) pasa a ser ln(x*a)
degree(4*x^5 − 3*x^4 + x*3 + x^2 − 4);	devuelve el grado del polinomio
coeff(x^2 − 3*x + 2, **x^2**); coeff(x^2 − 3*x + 2, x, 2);	extrae el coeficiente con grado del término indicado
coeffs(x^2 − 3*x + 2);	extrae una sucesión con los coeficientes
quo(x^3 − 3*x^2 + 2, x^2 − x + 1, x);	cociente de la división polinómica
rem(x^3 − 3*x^2 + 2, x^2 − x + 1, x);	resto de la división polinómica
numer((x + 1)/(x^2 − 3*x − x + 1));	extrae el numerador
denom((x + 1)/(x^2 − 3*x − x + 1));	extrae el denominador
convert(sin(x), tan);	nos da el seno en función de la tangente
convert(cos(x), tan);	nos da el coseno en función de la tangente
convert(arcsin(x), ln);	nos da el arco(seno(x)) en función del ln
convert(binomial(m, 3), factorial);	expresa el número combinatorio como cociente de factoriales
convert(1/2, float);	pasa el racional 1/2 a punto flotante
convert(taylor(e^x, x = 0, 10), polynom);	convierte serie en polinomio
assume(y < 0); assume(x > 5);	establece restricciones para las variables x e y, asociado a las órdenes **is** y **isgiven** puede resultar muy útil, por ejemplo si preguntamos:
Y;	la respuesta será: y~ (variable con restricción)
is(x*y > 0);	la respuesta será: false
Y:= 'y';	sirve para quitar la restricción impuesta

Tipos, operandos, argumentos

whattype(−3);	Averigua el tipo, en este caso: integer
whattype(1/2);	nos dice que es racional
type(3, integer);	devuelve el valor true, porque es entero
type(1/2, integer);	devuelve el valor false, porque es racional
op(j, expresión);	Extrae el operando j-ésimo de expresión
nops(expresión);	número de operandos de la expresión
args(i);	Dentro de un procedimiento devuelve el argumento i-ésimo
nargs( );	Dentro de un procedimiento devuelve el número de argumentos
lhs(3*x = ln(x) + 1);	lado izquierdo de una igualdad: 3*x
rhs(3*x = ln(x) + 1);	lado derecho de una igualdad: ln(x) + 1

Estructuras de control

if *condición* **then** *bloque1* ***else*** *bloque2* **fi**;	no es necesaria la cláusula else, aunque si debe terminar con fi;
if *condición1* **then** *bloque1* **elif** *condición2* then *bloque2* **elif** *condición3* **then** *bloque3* ..... **fi**;	otra forma del if, útil para evitar los if anidados cuando hay más de dos alternativas
for *variable* **from** *valor1* **to** *valor2* **do** *bloque* **od**;	se puede obviar: from *valor1,* si *valor1* = 1, y quedaría: for i to *valor2* do, también admite: by m, para saltos de m en m, etc.,.. break; permite salir del bucle sin terminarlo next; salta al siguiente valor de variable, otra forma útil es: for n in [3, 7, 9, 11]
while *condición* **do** *bloque* ***od;***	*mientras la condición se cumple se ejecuta el bloque*
bloque	puede ser cualquier número de sentencias separadas por puntos y comas
condición	relación lógica con conectivas: <, <=, =, >=, and, or, not. Ejemplos: x < = 3, modp(n, 2) = 0, not(x = y), x = 1 and y = 2, x^2 + y^2 = 0, etc,. etc,.

Procedimientos

nombre:= **proc(x, y, z, u)** local v1, v2, ...vn; *bloque*; **end**;	Los procedimientos tienen básicamente la estructura señalada, v1, v2, etc., son variables locales, x, y, z, u son los argumentos que le pasamos, la salida es bien con un RETURN(salida) o simplemente escribiendo el nombre de la variable resultado
ERROR(` se introdujo una lista vacía`);	lanza nuestros mensajes "de error" desde dentro de un procedimiento, debe ponerse tras un if que verifique la condición para la que se desea generar el "error"

Ejemplo 1º

gota:= proc(x, y);
if x^2 + y^2 = 0 then RETURN(1); else RETURN(sin(x^2 + y^2)/(x^2 + y^2));fi;
end;
	Una función de dos variables puede representarse como una superficie ,escribir: plot3d(gota(x, y), x = −5..5, y = −5..5);
	¡Ojo RETURN() se escribe siempre con mayúscula!

Ejemplo 2º

tangente:= proc(x, a, f)	le damos nombre y tendrá 3 argumentos, devolverá la ecuación de la tangente a f(x) en x = a
local fa, fp;	fa, fp son variables locales
#calcula la ecuación de la tangente a la función f en el	comentarios para documentar el procedimiento

punto (a, f(a)) #	
fa:= subs(x = a, f(x));	asignamos a fa el resultado de substituir x por a en f(x)
fp:= subs(x = a, diff(f(x), x));	asignamos a fp el valor de la derivada en x=a supuesta derivable
RETURN((x − a)*fp + fa)	devolvemos la ecuación pedida
end;	fin de procedimiento

Para hacerlo funcionar debemos definir una función e invocarlo:

F:= x->x^3;	definición de la función sobre la que trabajaremos, puede ser cualquier otra
tangente(x, 2, f);	invocación del procedimiento, le pasamos f, y los valores concretos a los que aplicaremos el procedimiento
plot({f(x), tangente(x, 2, f)}, x = −2..3);	dibujamos la curva y su tangente en x = 2

Lectura y escritura: entrada y salida de procedimientos

print(area);	sirve para visualizar las ordenes del procedimiento área
save(area, areafic);	salvamos el procedimiento área en memoria en un fichero dos que se llamará areafic y que quedará escrito en ascii puro
save (area, `areafic. m`);	salvamos el procedimiento en un fichero con formato interno **.m** de maple, luego podrá ser almacenado en una librería maple.lib, mediante el programa externo **march**.exe, que se halla en maplev3\bin
read rutina1;	leemos un fichero ascii llamado rutina1 con un procedimiento o con instrucciones maple (simplemente puestas una a continuación de otra) situado en el directorio de trabajo
read `c:/progs/rutina.txt`	leemos un fichero ascii situado en c:\progs llamado rutina.txt (ojo, la barra de subdirectorio, en Maple, es la tecla del 7)
libname;	variable que almacena los caminos de búsqueda de librerías, es un simple string modificable, admite, separados por comas, otros posibles path
Interface (verboseproc = 2);	con esta orden podremos visualizar el contenido de procedimientos escritos en maple y que sean accesibles
writeto(salida);	con esta orden toda la salida del maple pasa a un fichero del dos llamado salida y situado en el directorio de trabajo
print(sin);	las ordenes del procedimiento sin(x) irán a parar al fichero salida, si antes se ejecutaron las ordenes previas
writeto(terminal);	redirige la salida al terminal
interface(verboseproc=1);	ponemos el nivel de visualización de los procedimientos en su defecto